Herausforderung Multikern-Systeme

Walter Tichy, Victor Pankratius

Universität Karlsruhe (TH)
Institut für Programmstrukturen
und Datenorganisation
Am Fasanengarten 5
76131 Karlsruhe

{tichy|pankratius}@ipd.uni-karlsruhe.de

Abstract: Multikern-Prozessoren stellen die Softwaretechnik vor die Herausforderung, leistungshungrige Anwendungen aller Art zu parallelisieren. Bereits heute bieten handelsübliche Chips bis zu 64-fache Parallelität, und eine Verdopplung der Prozessorzahl wird für jede neue Chip-Generation vorhergesagt. Da die Taktfrequenzen nicht mehr wesentlich steigen werden, müssen Leistungssteigerungen über Parallelisierung erreicht werden. Hierzu werden neue Konzepte und Werkzege benötigt, damit Parallelisierung in die Routinetätigkeit des Softwaretechnikers integriert werden kann. Paralleles Rechnen ist bereits im Laptop, PC, Server und PDA/Telefon angekommen und wird demnächst auch bei eingebetteten Systemen wichtig werden. Wir erläutern die mittelfristig wichtigen Forschungsaufgaben aus unserer Sicht, gegründet auf unserer Erfahrung mit parallelem Rechnen.

1 Aktuelle Situation

In der Informatik kündigt sich derzeit ein klassischer Paradigmenwechsel an: Der Übergang vom sequenziellen zum parallelen Rechnen auf breiter Front. Während die Parallelverarbeitung im ersten halben Jahrhundert der Informatik auf wenige Anwendungsbereiche beschränkt war (wissenschaftliches Rechnen, Datenbanken, Parallelismus auf Instruktionsebene), wird nun mit Multikern-Prozessorchips die Parallelverarbeitung für jeden erschwinglich[1] und dadurch in einem breiten Anwendungsspektrum möglich. Verstärkt wird diese Entwicklung dadurch, dass die Taktraten der Prozessoren seit 2002 nicht mehr wesentlich gestiegen sind (siehe Abb. 1). Ein fortgesetzter exponentieller Anstieg der Taktraten ist wegen der Hitzeentwicklung ausgeschlossen, selbst wenn diverse Techniken zur Reduzierung des Energieverbrauchs entwickelt werden. Die Folgerung daraus ist, dass zukünftige Leistungssteigerungen im Wesentlichen über Parallelisierung erreicht werden müssen.
Glücklicherweise ist die exponentielle Steigerung der Integrationsdichte ungebrochen, so

[1] Im Herbst 2007 war der Preis für ein Doppelprozessorchip von 500 auf 50 Euro gefallen. Doppelprozessorchips werden sogar in Laptops eingebaut.

dass David Patterson von Berkeley eine neue Version der Moore'schen Regel vorgeschlagen hat: Eine Verdopplung der Anzahl Prozessoren mit jeder Chipgeneration, bei etwa gleicher Taktfrequenz [ABC+06].

Wieviele Prozessoren passen auf ein Chip? Bereits 2005 kamen IBMs Cell mit 9 Prozessoren und SUNs Niagara mit 8 Prozessoren und 32 Fäden pro Chip heraus. Zwei Jahre später schon verdoppelte SUN die Anzahl der Ausführungseinheiten und Fäden auf dem Niagara2 Chip und fügte leistungsfähige Gleitkommaverarbeitung hinzu. Auch Cell dürfte bald mehr Prozessoren aufweisen. Der derzeitige Rekord ist aber wesentlich weiter voraus: 2005 entwickelte Cisco fast unbemerkt den Metro Chip mit 192(!) Prozessoren auf 3,24 cm^2[Eat05]. Wenn man statt der damaligen 130 nm Technologie aktuellere 45 nm Technologie einsetzen würde, könnte man 1500 dieser 32-Bit Prozessoren auf einem Chip unterbringen. Glaubt man den Vorhersagen der Halbleiter-Industrie, dann könnte es bis 2017 10^{11} Transistoren auf einem Chip geben. Da ein Festkommaprozessor sowie eine Gleitkommaeinheit jeweils ca. 10^5 Transistoren benötigen, könnte man, wenn man nur 10 Prozent des Chips für Prozessoren benutzt (der Rest geht an Verbindungsnetze und Caches), in zehn Jahren in etwa 100.000 Prozessoren auf einem Chip unterbringen.

Offensichtlich ist es höchste Zeit, sich auf diese Entwicklung einzustellen – auch in der Softwaretechnik. Wenn es nicht gelingt, die durch Parallelität weiter steigende Rechenleistung in leistungsfähiger Software zu nutzen, wäre das gleichbedeutend mit einer zweiten Softwarekrise.

Leider ist die Softwaretechnik außer bei numerischen Anwendungen schlecht für massive Parallelität gerüstet. Das Problem ist ein Mangel an Know-How, wie man parallele Anwendungen konstruiert. Angesichts der sich eröffnenden Möglichkeiten besteht ein dringender Bedarf, brauchbare Methoden, Konzepte und Werkzeuge zu entwickeln, die es jedem Softwareentwickler ermöglichen, korrekte und effizient ausführbare parallele Programme systematisch zu erstellen. Um die Potenziale moderner Multikern-Rechner ausschöpfen zu können, müssen möglicherweise alle Bereiche der Softwaretechnik im Lichte der aktuellen Entwicklungen überdacht werden. Dieser Artikel identifiziert die aus unserer Sicht wichtigsten Gebiete, die in der Softwareforschung mittelfristig angegangen werden müssen, um die Parallelprogrammierung für den Alltag tauglich zu machen. Wir gründen unsere Sicht auf die jahrzehntelange Erfahrung der Forschungsgemeinde mit parallelen Systemen, angefangen mit CMUs C.mmp Mitte der 70er Jahre, über die Connection Machine (Mitte der 80er) bis hin zu unseren Arbeiten mit Clustern und deren Programmierung (Mitte 90er), Studien der gegenwärtigen Situation (z.B. [ABC+06]), Diskussionen mit Herstellern und Hardware-Architekten sowie Workshops zu diesem Thema.

2 Die Herausforderungen in der Softwaretechnik

Die Softwaretechnik kann aus dem langjährigen Einsatz der Parallelität im Bereich des wissenschaftliches Rechnens sehr viel lernen. Aber das allein wird nicht reichen! Verglichen mit den großen Anwendungen auf PCs und Servern, die in die Millionen Zeilen Code gehen, sind numerische Anwendungen klein und arbeiten mit einer überschaubaren

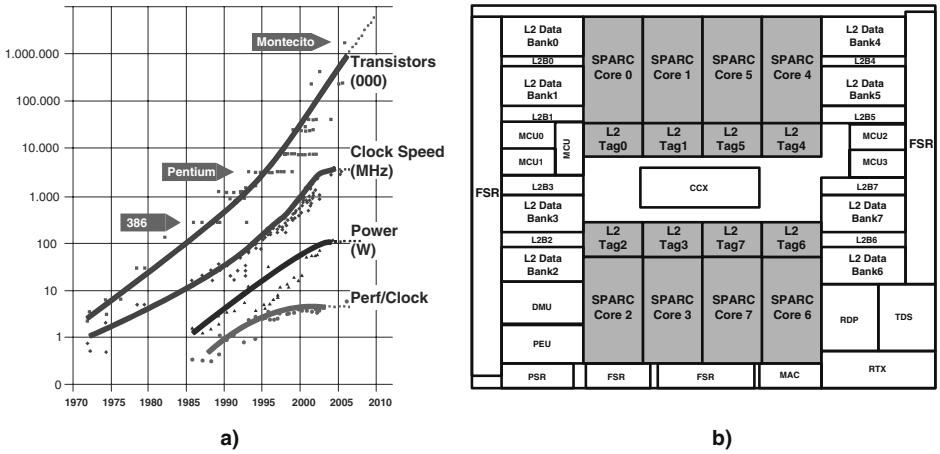

Abbildung 1: a) Trends in der Hardware-Entwicklung [Smi07]; b) Architekturskizze des SUN Niagara2 Prozessors mit 8 Kernen.

Menge von Datenstrukturen wie Vektoren, Matrizen und einigen unregelmäßigen Gebietszerlegungen. Dagegen werden die funktionsreichen Anwendungen auf PCs und Servern sehr viel mehr Möglichkeiten zur Parallelisierung bieten, womöglich auf mehreren Ebenen gleichzeitig. Ein weiterer, wichtiger Unterschied ist, dass die Entwickler wissenschaftlicher Software meist auch die einzigen Benutzer dieser Software sind. Entsprechend sind Qualitäten wie Benutzerfreundlichkeit, Zuverlässigkeit, Robustheit oder Wartbarkeit dort weniger wichtig als für Software, die von Tausenden oder Millionen von Menschen eingesetzt wird, eine Lebensdauer von Jahrzehnten hat, unternehmenskritische Daten verwaltet, oder sicherheitskritische Anlagen steuert. Die Herausforderung für die Softwaretechnik ist daher, die Parallelisierung von großen Anwendungen bei wesentlich höheren Komplexitäts- und Qualitätsanforderungen zu meistern.

Eine negative Erfahrung aus dem Bereich des wissenschaftlichen Rechnens ist bedeutsam: Die automatische Parallelisierung funktioniert schon für relative einfache, numerische Codes nicht zufriedenstellend [ABC$^+$06]. Bei komplexen, nicht-numerischen Anwendungen wird es noch deutlich schwieriger werden. Der Grund ist verständlich: Die automatische Parallelisierung entspricht der Herleitung eines parallelen Algorithmus für ein Problem, für das nur eine sequenzielle Implementierung, und damit nicht einmal eine Spezifikation, vorliegt. Schon die automatische Herleitung sequenzieller Algorithmen aus präzisen Spezifikationen ist nicht praktikabel; wie sollte es dann für parallele Algorithmen klappen? Softwaretechniker werden wohl auf absehbare Zeit die Parallelisierung nicht an Automaten delegieren können.

Für die manuelle Erstellung paralleler Systeme sind folgende Themen mittelfristig wichtig: Programmiersprachen und -modelle; Synchronisation; Autotuning; Zuverlässigkeit; Reengineering. Ferner müssen Konzepte der parallelen Programmierung und Algorithmen früh in die Ausbildung einfließen. Eine wissenschaftliche Gemeinde für die Softwaretech-

nikfragen von allgemeinen, parallelen Anwendungen könnte den Erkenntnisaustausch und damit den Fortschritt beschleunigen. Wir betrachten nun jedes dieser Themen genauer.

3 Programmiersprachen und -modelle

Bei der Programmierung von Multikern-Rechnern stellt sich die grundlegende Frage, auf welche Weise Entwickler mit Parallelität konfrontiert werden. Zurzeit gibt es mehrere Ansätze mit unterschiedlichen Abstraktionsmöglichkeiten, von denen wir einige beispielhaft erwähnen.

In C++ gibt es keine nativen Konstrukte für Parallelität. Hierfür werden Erweiterungen wie POSIX Threads benötigt. Im Vergleich dazu hat Java zwar native Sprachkonstrukte für die Erzeugung/Zerstörung von Fäden sowie zur Synchronisation, jedoch befinden sich diese größtenteils auf einer niedrigen Abstraktionsbene, bei der sich Entwickler um alle Details genauestens kümmern muss. Diese Art der parallelen Programmierung ist fehleranfällig.

Parallele Bibliotheken, wie z.B. Intel Threading Building Blocks, Intel Math Kernel Library oder AMD Core Math Library, verstecken die Parallelität in Bibliotheken, die aus sequenziellen Programmen aufgerufen werden. Obwohl Bibliotheken ein wichtiger Schritt in Richtung Parallelisierung sind, können sie oft nur feingranulare Parallelität ausnutzen. Außerdem müssen diese Bibliotheken auch geschrieben werden.

OpenMP ist eine verbreitete Direktivensprache, die in eine andere Wirtssprache, wie z.B. C/C++ oder Fortran, eingebettet wird. OpenMP besitzt ein Fork-Join-Modell, in dem durch Direktiven implizit mehrere Fäden erzeugt werden, die eine Aufgabe parallel bearbeiten. Insbesondere bietet OpenMP die bekannte, asynchrone Forall-Anweisung [TPHL92], die unabhängige Iterationen einer Schleife parallel ausführt. Ähnliche Konstrukte gibt es inzwischen auch in den Microsoft Parallel Extensions für .NET. Positiv hervorzuheben ist, dass OpenMP-Entwickler von Routineaufgaben der parallelen Programmierung auf niedriger Abstraktionsebene befreit werden. Schwierig ist noch das Debugging von OpenMP-Programmen, da sich eine Wirtssprache der Direktiven-Erweiterungen in der Regel nicht bewusst ist. Eine Integration der Parallelisierungskonstrukte in eine sequenzielle Programmiersprache ist aber durchaus denkbar.

Des Weiteren gibt es Ansätze, die domänenspezifische Konstrukte für Parallelität zur Verfügung stellen und auch grobkörnigen Parallelismus umfassen. In sogenannten Strom-Programmiersprachen wie StreamIt [GTA06] besteht ein Programm aus einer Menge von Filtern, die über dedizierte Datenkanäle verbunden sind und Datenströme parallel bearbeiten. Dieses Programmiermodell wird insbesondere im Audio-/Video-Bereich und bei der Signalverarbeitung benutzt. Ein anderes Beispiel ist die Sprache ZPL [CCL+00], die insbesondere auf Feld-Datenstrukturen arbeitet. Sie bietet mächtige Konstrukte zur Auswahl und Manipulation von Daten in Feldern und ist insbesondere für Simulationen, wie z.B. das N-Körper-Problem oder Klimasimulationen, geeignet.

Die Herausforderungen für die Softwaretechnik. Es fehlt noch ein einheitlicher Ansatz zur Programmierung von Multikern-Rechnern, der verschiedene Aspekte der dargestellten Programmiermodelle vereinigt. Welches Programmiermodell oder welche Kom-

bination von existierenden Programmiermodellen ist am besten geeignet? Welche Konstrukte sind für die Entwicklung allgemeiner parallele Software notwendig und sinnvoll? Wie können unterschiedliche Abstraktionsebenen der Parallelität unterstützt werden? Wie soll abgewogen werden zwischen maschinenspezifischen Details/Performanz vs. Portabilität/Wartbarkeit?

4 Synchronisation

Synchronisation ist ein inhärentes Problem der parallelen Programmierung und wird insbesondere dort benötigt, wo gleichzeitig auf gemeinsam genutzte Daten zugegriffen werden kann. Derzeit überwiegt ein Synchronisationsmodell, in dem für die Koordination der parallelen Zugriffe explizite Sperren benutzt werden. Üblicherweise liegt es in der Verantwortung des Programmierers, die Sperren an den richtigen Stellen zu setzen und das zugehörige Zugriffsprotokoll korrekt umzusetzen. Diese Vorgehensweise ist fehleranfällig und führt zu Ausfällen, deren Ursachen wegen nicht-deterministischer Abläufe schwer einzukreisen sind. Typische Probleme (z.B. Wettlaufsituationen, Verklemmungen, geschachtelte Monitore) sowie Methoden für die automatische Defektdetektion werden in [Ott07] beschrieben. Ein weiteres Problem betrifft die Granularität der Sperren: Feingranulare Sperren sind performanter, aber schwieriger zu programmieren; grobgranulare Sperren sind einfacher, führen jedoch häufig zu schlechter Performanz.

Transaktionaler Speicher (engl. transactional memory) ist ein jüngerer Ansatz, der bei der Synchronisation auf Sperren verzichtet und die damit verbundenen Probleme umgeht (vgl. [LR07] für einen Überblick). Ähnlich wie im Datenbankbereich ist die grundlegende Idee dabei die Zusammenfassung einer Folge von Programmanweisungen zu einer atomaren Transaktion, die entweder ganz oder gar nicht ausgeführt wird. Atomizität kann mit unteilbaren Befehlen wie z.B. compare-and-swap realisiert werden. Vor und nach einer Transaktion ist garantiert, dass der Speicher in einem konsistenten Zustand ist. Weiterhin wird eine Transaktion isoliert ausgeführt, d.h. dass es aus Sicht einer Transaktion keine anderen Transaktionen gibt und auch keine Abhängigkeiten zwischen diesen bestehen.

Die Herausforderungen für die Softwaretechnik. Wie können Synchronisationsfehler automatisch detektiert werden? Auch bei transaktionalem Speicher können Verklemmungen auftreten. Weitere Probleme sind die Performanz transaktionalen Speichers sowie die häufige Annahme, dass Transaktionen verhältnismäßig klein und die Kosten für das Rücksetzen gering sind. Hier ist weitere Forschung nötig: Wie kann die Performanz verbessert werden? Welche Hardware-Unterstützung ist nötig? Wie kann transaktionaler Speicher in die Programmiermodelle für Multikern-Rechner integriert werden? Gibt es für Synchronisation andere Alternativen?

5 Autotuning

Bei der Parallelisierung sind eine Reihe von Parametern zu optimieren, wie die Anzahl der eingesetzten Fäden, die Aufteilung der Arbeit, die Größe der Datenstrukturen, die Sperrgranularität, die Aufgabenzuweisung bei heterogenen Prozessoren, usw. Diese Parameter sind schwierig zu ermitteln und zudem von Plattform zu Plattform unterschiedlich. Eine automatische Bestimmung ist daher zwingend notwendig.

Autotuning ist ein Ansatz, bei dem die optimalen Ausführungsparameter für ein paralleles Programm durch systematische und automatische Messungen ermittelt werden (vgl. [WKT00, ABC+06]). Typische Zielfunktionen zur Optimierung sind Ausführungszeit, Energieverbrauch oder Genauigkeit der Ergebnisse. Das Autotuning geschieht typischerweise auf der Zielumgebung entweder in einer Phase vor oder während der eigentlichen Programmausführung.

Um Autotuning zu ermöglichen, müssen parallele Programme parametrisierbar sein. Parallelität kann außerdem auf unterschiedlichen Abstraktionsebenen ausgedrückt werden. Auf einer niedrigen Ebene kann beispielsweise die Anzahl zu verwendender Fäden durch eine Variable ausgedrückt werden, deren optimaler Wert vom Autotuner bestimmt wird. Auf einer höheren Ebene können konfigurierbare, parallele Entwurfsmuster benutzt werden. Ein Beispiel dafür ist das Muster einer Pipeline, die in mehreren Stufen Berechnungen parallel durchführt und deren Stufenzahl dynamisch variiert werden kann. Erste Experimente mit der Parallelisierung einer kommerziellen Messtechnik-Anwendung (vgl. Abb. 2) haben gezeigt, dass schon durch die alleinige Verwendung derartiger Entwurfsmuster, ohne Modifikation der zugrunde liegenden Algorithmen, Speedups von 2.9 erreicht werden konnten [PSJT07].

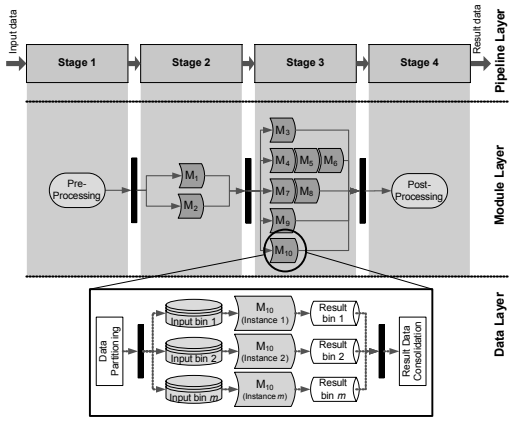

Abbildung 2: Beispiel für Parallelität auf verschiedenen Abstraktionsebenen [PSJT07].

Autotuner werden eine wichtige Position bei Multikern-Anwendungen einnehmen, da sich jetzt schon herauskristallisiert, dass viele Optimierungen nicht alleine den Übersetzern überlassen werden können [ABC+06]. Ein Grund dafür ist, dass Übersetzer eine große Zahl an Optimierungsmöglichkeiten handhaben müssen, diese aber nicht notwendigerwei-

se aus dem Programmcode ersichtlich sind. Im Gegensatz dazu könnte beim Autotuning über Entwurfsmuster zusätzliches Wissen des Entwicklers in die Optimierung mit einfließen.

Die Herausforderungen für die Softwaretechnik. Obwohl rudimentäre Autotuning-Ansätze im Bereich der Numerik schon vorhanden sind [WPD01, ABC+06], gibt es viele offene Fragen in Bezug auf allgemeine parallele Software: Wie kann der Parameterraum für einen Autotuner reduziert werden? Können dabei Methoden aus der Programmanalyse helfen? Wie sehen geeignete Vorhersagemodelle aus? Wie kann die Konfigurierbarkeit allgemeiner paralleler Programme elegant beschrieben werden? Was sind typische Entwurfsmuster für allgemeine parallele Programme und wie können Konfigurationsmöglichkeiten in den Mustern ausgedrückt werden? Wie soll das Zusammenspiel zwischen Übersetzer-Optimierungen und Autotuning aussehen?

6 Ausfallsicherheit

Neben Performanz ist Ausfallsicherheit der Software ein weiteres wichtiges Anwendungsgebiet für Multikern-Prozessoren. Im Gegensatz zu früheren Ansätzen, wie z.B. beim Tandem-Computer, stellen die mehrfach vorhandenen Kerne kostengünstige Redundanz dar. Dies eröffnet für Standard-Software ebenfalls neue Möglichkeiten, durch redundante Programmausführung die Ausfallsicherheit zu erhöhen und eine rasche (möglicherweise sogar transparente) Erholung von Systemabstürzen zu ermöglichen.

Die Herausforderungen für die Softwaretechnik. Wie sehen Methoden der Ausfallsicherheit für Multikern-Rechner aus? Was kann man aus dem Bereich der Fehlertoleranz übertragen? Auf welche Weise sollen diese Methoden in allgemeiner Software integriert werden? Wie werden Virtualisierungstechniken eingesetzt und wie sehen zukünftige Systemarchitekturen dann aus?

7 Eingebettete Systeme

Multikern-Architekturen sind im Bereich eingebetteter Systeme nicht nur aus Gründen der erhöhten Leistung und Ausfallsicherheit attraktiv. Beispielsweise befinden sich in modernen Autos eine Vielzahl von Prozessoren und mehrere Bus-Systeme; Elektronik und Software machen etwa 40% der Herstellungskosten aus [Bro06]. Multikern-Prozessoren könnten durch eine Re-Zentralisierung zu einer Kostenreduktion führen, indem verschiedene Kerne die Funktionen der jetzt getrennt realisierten Prozessoren übernehmen. Weiterhin könnten Verbindungen eingespart und Latenzzeiten verkürzt werden.

Die Herausforderungen für die Softwaretechnik. Um diese Möglichkeiten ausschöpfen zu können, ist weitere Forschung in der Softwaretechnik für eingebettete Systeme notwendig, insbesondere unter Berücksichtigung von Echtzeitbedingugen.

8 Reengineering

Viele der im Alltag verwendeten Anwendungen sind explizit für sequenzielle Hardware konzipiert. Oft können sie aus ökonomischen Gründen nicht von Grund auf neu parallel programmiert werden. Daher stellt sich die Frage, wie man mit Reengineering-Methoden sequenzielle Programme parallelisiert, um die Potenziale der Multikern-Hardware ausnutzen zu können.

In der Vergangenheit hat sich bereits gezeigt, dass eine automatische Parallelisierung selbst für spezielle Domänen kaum erfolgreich ist. Interaktive Transformations-Werkzeuge, die dem Entwickler Routinearbeiten abnehmen, versprechen im Multikern-Kontext mehr Erfolg. Insbesondere können beim Refaktorisieren sequenzieller Anwendungen parallele Entwurfsmuster (vgl. Abschnitt 5) verwendet werden.

Weiterhin müssen Methoden des Programmverstehens, Programmvisualisierung und Programmanalyse für parallele Programme vorangetrieben werden, da in Zukunft die „Altprogramme" parallel sein werden.

Die Herausforderungen für die Softwaretechnik. Wie können vorhandene sequenzielle Programme in parallele Programme transformiert werden? Welche Routineaufgaben der Entwickler können automatisiert werden? Wie können welche parallelen Entwurfsmuster beim Refaktorisieren eingepflanzt werden? Wie müssen Methoden des Programmverstehens für allgemeine parallele Software aussehen?

9 Lehre

Alle Bereiche der Informatikausbildung müssen im Lichte der Paralellverarbeitung neu überdacht werden. Während derzeit die Sequenzialität der Normalfall ist, könnte in Zukunft die Parallelität zum Normalfall werden.

Die Studierenden und Auszubildenden von heute sind die Softwareentwickler von morgen, die Software in allen Bereichen für Multikern-Plattformen entwickeln werden. Um international konkurrenzfähig zu bleiben, muss in allen Ausbildungsstätten rechtzeitig dafür gesorgt werden, dass die Entwickler die entsprechenden Fähigkeiten besitzen – Parallelität muss daher in jedem Studienplan vorhanden sein.

10 Bildung einer wissenschaftlichen Gemeinde

Für eine Forschung im Multikern-Bereich ist eine organisierte wissenschaftliche Gemeinde notwendig. In der Gesellschaft für Informatik bietet der Arbeitskreis *Software Engineering für parallele Systeme (SEPAS)* [GI-07] einen entsprechenden Rahmen, der Forscher und Praktiker zusammenbringt und den Austausch intensiviert.

Ein Ziel des Arbeitskreises ist, dass die im Alltag eingesetzte Software das Potenzial mo-

derner Multikern-Rechner ausschöpft. Weiterhin sollen aus der Kollaboration von Forschern und Praktikern neue Impulse für die Forschung gewonnen werden, siehe http://www.multicore-systems.org/gi-ak-sepas

11 Zusammenfassung und Ausblick

Die Softwaretechnik, wie auch die Informatik als Ganzes, befindet sich derzeit an einem Wendepunkt. Parallele Hardware ist in Form von Multikern-Rechnern für jedermann erschwinglich geworden. Neue, anspruchsvolle Anwendungen könnten die Fähigkeiten dieser Hardware nutzen, zum Beispiel in Form von intelligenteren Funktionen, genaueren Ergebnissen oder schnelleren Antworten. Allerdings fehlen auf der Seite der Softwaretechnik alltagstaugliche Konzepte und Methoden, um über das wissenschaftliche Rechnen hinaus Multikern-Software zu entwickeln.

Die Gemeinde der Softwaretechniker – und auch Sie, werter Leser – haben nun die seltene, ja vielleicht einmalige Chance, in einer neuen Pionierzeit der Informatik aktiv zu werden und die Zukunft maßgeblich mit zu gestalten.

Danksagung

Wir bedanken uns bei den Mitgliedern der Karlsruher Gruppe *Software Engineering für Multikern-Systeme*, Benjamin Herd, Ali Jannesari, Frank Otto, Christoph Schaefer und Wolfgang Schnerring für ihre Unterstützung.

Literatur

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams und Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Bericht UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 18. Dezember 2006.

[Bro06] Manfred Broy. Challenges in automotive software engineering. In *ICSE '06: Proc. of the 28th international conference on Software engineering*, Seiten 33–42, New York, NY, USA, 2006. ACM.

[CCL+00] B.L. Chamberlain, Sung-Eun Choi, C. Lewis, C. Lin, L. Snyder und W.D. Weathersby. ZPL: a machine independent programming language for parallel computers. *Transactions on Software Engineering*, 26(3):197–211, 2000.

[Eat05] Will Eatherton. The Push of Network Processing to the Top of the Pyramid. Symposium on Architectures for Networking and Communications Systems, 26.–28. Oktober 2005. http://www.cesr.ncsu.edu/ancs/slides/eathertonKeynote.pdf, letzter Abruf 17.12.2007.

[GI-07] GI-Arbeitskreis Software Engineering für parallele Systeme (SEPAS). http://www.multicore-systems.org/gi-ak-sepas, 2007.

[GTA06] Michael I. Gordon, William Thies und Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th international conference on architectural support for programming languages and operating systems (ASPLOS-XII)*, Seiten 151–162, New York, NY, USA, 2006. ACM Press.

[LR07] James R. Larus und Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2007.

[Ott07] Frank Otto. Analyse von Java-Programmen auf Synchronisierungsfehler. Diplomarbeit, Institut für Programmstrukturen und Datenorganisation (IPD), Universität Karlsruhe (TH), 2007.

[PSJT07] Victor Pankratius, Christoph Schaefer, Ali Jannesari und Walter F. Tichy. Software Engineering for Multicore Systems–An Experience Report. Technischer Bericht, Universität Karlsruhe (TH), Dezember 2007.

[Smi07] Burton Smith. Reinventing Computing. Manycore Computing Workshop, June 20–21 2007.

[TPHL92] Walter F. Tichy, Michael Philippsen, Ernst A. Heinz und Paul Lukowicz. From Modula-2* to Efficient Parallel Code. In *3rd Workshop on Compilers for Parallel Computers*, Jgg. 2, Seiten 186–200, Wien, Österreich, 1992.

[WKT00] Otilia Werner-Kytölä und Walter F. Tichy. Self-Tuning Parallelism. In *Proc. 8th International Conference High Performance Computing and Networking (HPCN Europe)*, Jgg. 1823 of *LNCS*, Seiten 300–312, Amsterdam, The Netherlands, May 2000. Springer Verlag.

[WPD01] R. Clint Whaley, Antoine Petitet und Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, January 2001.